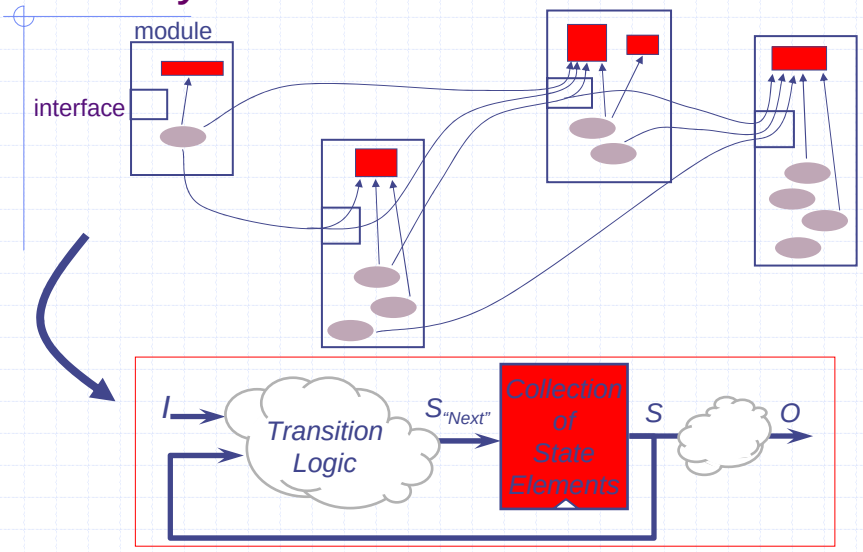
Bluespec-4: Rule Scheduling and Synthesis

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

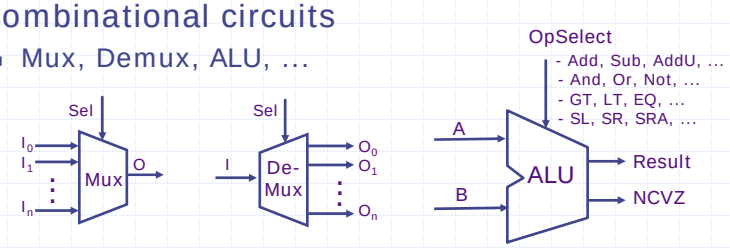
Based on material prepared by Bluespec Inc, January 2005

Synthesis: From State & Rules into Synchronous FSMs

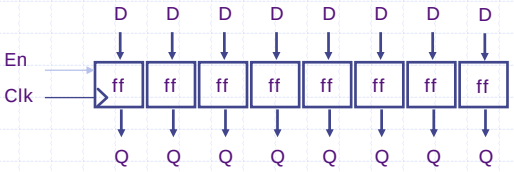


Hardware Elements

- Combinational circuits
  - Mux, Demux, ALU, ...

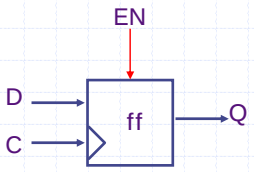


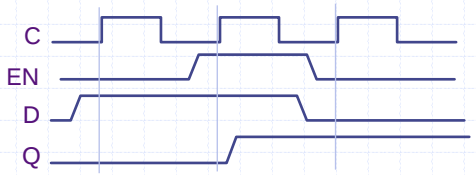
- Synchronous state elements
  - Flipflop, Register, Register file, SRAM, DRAM



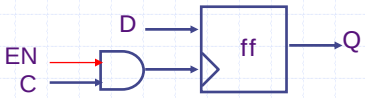
register

Flip-flops with Write Enables

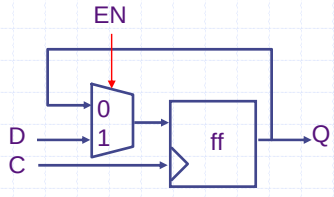




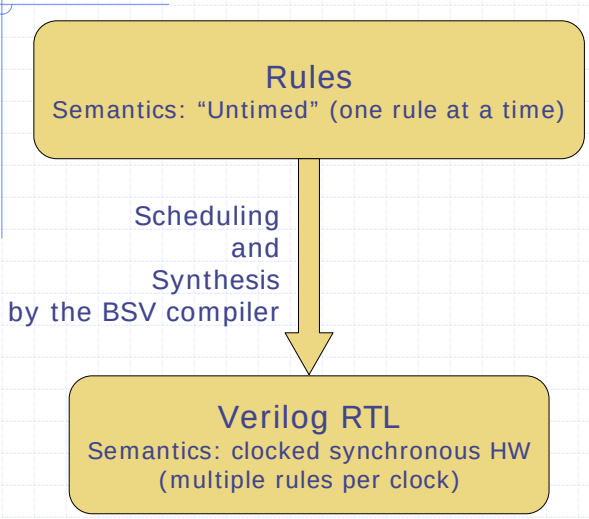
Edge-triggered: Data is sampled at the rising edge



dangerous!



Semantics and synthesis



Verification activities

Using Rule Semantics, establish functional correctness

Using Schedules, establish performance correctness

Rule semantics

Given a set of rules and an initial state

while (some rules are applicable*
in the current state)

- choose *one* applicable rule
- apply that rule to the current state to produce the next state of the system**

(*) "applicable" = a rule's condition is true in current state

(**) These rule semantics are "untimed" – the action to change the state can take as long as necessary provided the state change is seen as atomic, i.e., not divisible.

Why are these rule semantics useful?

- Much easier to reason about correctness of a system when you consider just one rule at a time
- No problems with concurrency (e.g., race conditions, mis-timing, inconsistent states)
  - We also say that rules are "interlocked"

➔ *Major impact on design entry time and on verification time*

Extensive supporting theory

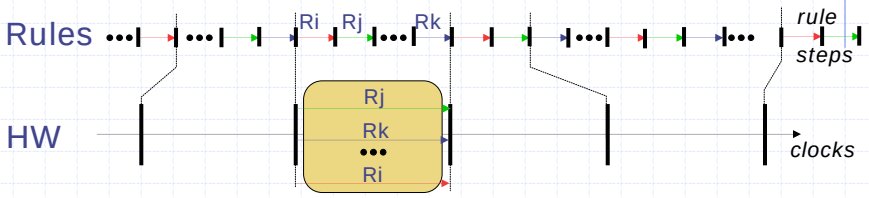
- *Term Rewriting Systems*, Terese, Cambridge Univ. Press, 2003, 884 pp.
- *Parallel Program Design: A Foundation*, K. Mani Chandy and Jayadev Misra, Addison Wesley, 1988
- *Using Term Rewriting Systems to Design and Verify Processors*, Arvind and Xiaowei Shen, IEEE Micro 19:3, 1998, p36-46
- *Proofs of Correctness of Cache-Coherence Protocols*, Stoy et al, in Formal Methods for Increasing Software Productivity, Berlin, Germany, 2001, Springer-Verlag LNCS 2021
- *Superscalar Processors via Automatic Microarchitecture Transformation*, Mieszko Lis, Masters thesis, Dept. of Electrical Eng. and Computer Science, MIT, 2000
- *... and more ...*

The intuitions underlying this theory are easy to use in practice

Bluespec's synthesis introduces concurrency

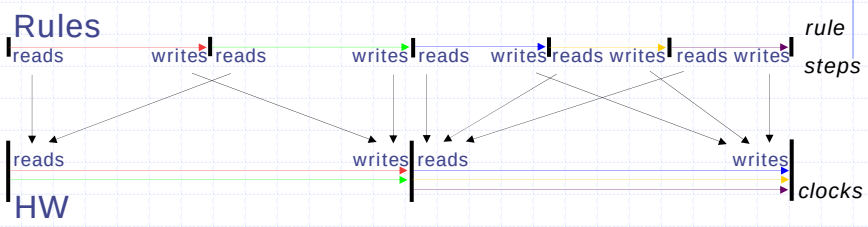
- Synthesis is all about executing multiple rules "simultaneously" (in the same clock cycle)
  - A. When executing a set of rules in a clock cycle in hardware, each rule reads state from the leading clock edge and sets state at the trailing clock edge
    - ⇒ none of the rules in the set can see the effects of any of the other rules in the set
  - B. However, in one-rule-at-a-time semantics , each rule sees the effects of all previous rule executions
- Thus, a set of rules can be *safely* executed together in a clock cycle only if A and B produce the same net state change

Pictorially



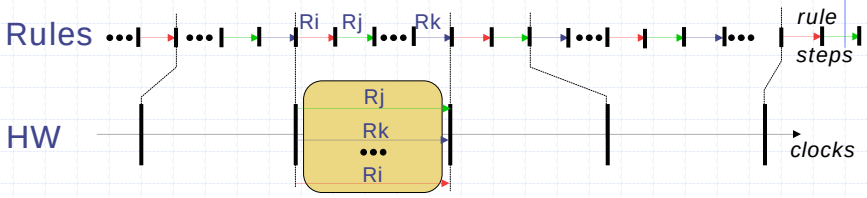
- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges
- In each clock, a different number of rules may fire

Parallel execution reorders reads and writes



- In the rule semantics, each rule sees (reads) the effects (writes) of previous rules
- In the HW, rules only see the effects from previous clocks, and only affect subsequent clocks

Correctness



- Rules are allowed to fire in parallel only if the net state change is equivalent to sequential rule exeuction
- Consequence: the HW can never reach a state unexpected in the rule semantics
- Therefore, correctness is preserved

Scheduling

- The tool schedules as many rules in a clock cycle as it can prove are safe
  - Generates interlock hardware to prevent execution of unsafe combinations
- Scheduling is the tool's best attempt at the fastest possible *correct* and *safe* hardware
  - *Delayed rule execution* is a consequence of safety checking, i.e., a rule "conflicts" with another rule in the same clock, the tool may delay its execution to a later clock

Obviously safe to execute simultaneously

```
rule r1; x <= x + 1; endrule
rule r2; y <= y + 2; endrule
```

```
always @(posedge CLK)
    x <= x + 1;
always @(posedge CLK)
    y <= y + 2;
```

```
always @(posedge CLK) begin
    x <= x + 1;
    y <= y + 2;
end
```

- Simultaneous execution is equivalent to r1 followed by r2
- And also to r2 followed by r1

Safe to execute simultaneously

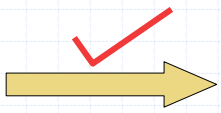
```
rule r1; x <= y + 1; endrule
rule r2; y <= y + 2; endrule
```

```
always @(posedge CLK)
    x <= y + 1;
always @(posedge CLK)
    y <= y + 2;
```

- Simultaneous execution is equivalent to r1 followed by r2
- *Not equivalent* to r2 followed by r1
  - But that's ok; just need equivalence to *some* rule sequence

Actions within a single rule

```
rule r1;
  x <= y + 1;
  y <= x + 2;
endrule
```



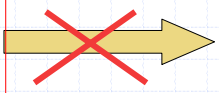
```
always @(posedge CLK)
    x <= y + 1;
always @(posedge CLK)
    y <= x + 2;
```

- Actions within a single rule *are* simultaneous

(The above translation is ok assuming no interlocks needed with any other rules involving x and y)

Not safe to execute simultaneously

```
rule r1;
  x <= y + 1;
endrule
rule r2;
  y <= x + 2;
endrule
```



```
always @(posedge CLK)
    x <= y + 1;
always @(posedge CLK)
    y <= x + 2;
```

- Simultaneous execution
  - is not equivalent to r1 followed by r2
  - nor to r2 followed by r1

A rule is not a Verilog "always" block! Interlocks will prevent these firing together (by delaying one of them)

Rule: As a State Transformer

A rule may be decomposed into two parts $\pi(s)$ and $\delta(s)$ such that

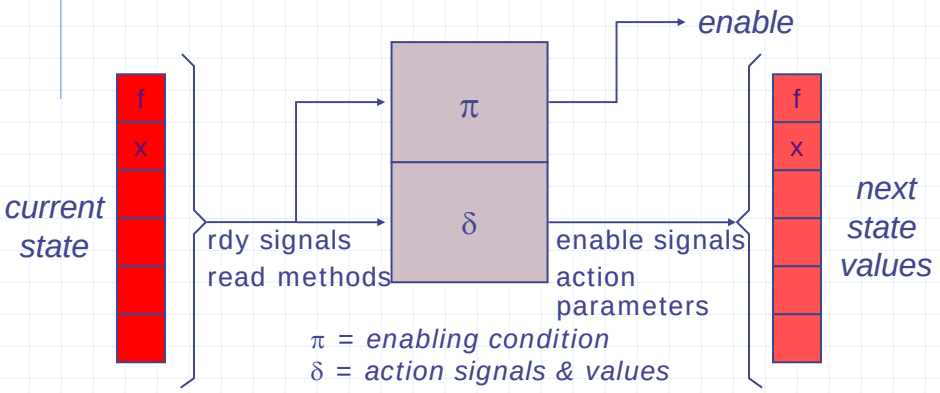
$$s_{next} = \textit{if } \pi(s) \textit{ then } \delta(s) \textit{ else } s$$

$\pi(s)$ is the condition (predicate) of the rule, a.k.a. the "CAN_FIRE" signal of the rule.
(conjunction of explicit and implicit conditions)

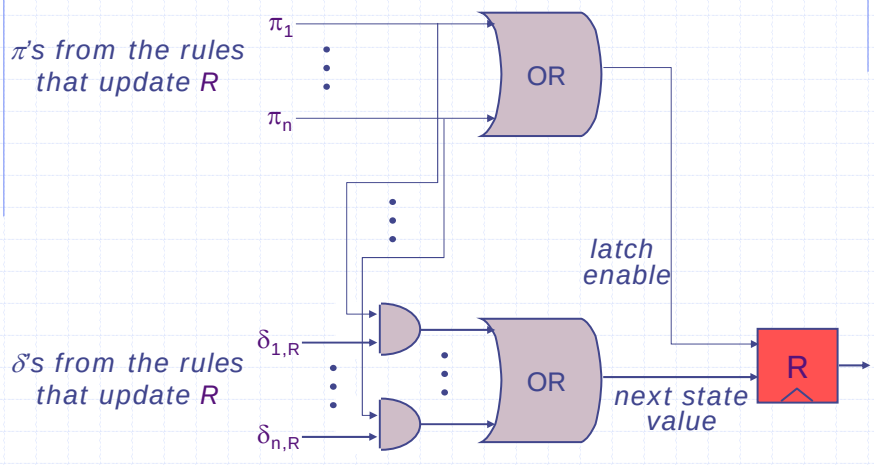
$\delta(s)$ is the "state transformation" function, i.e., computes the next-state value in terms of the current state values.

Compiling a Rule

```
rule r (f.first() > 0) ;
        x <= x + 1 ;    f.deq ();
endrule
```

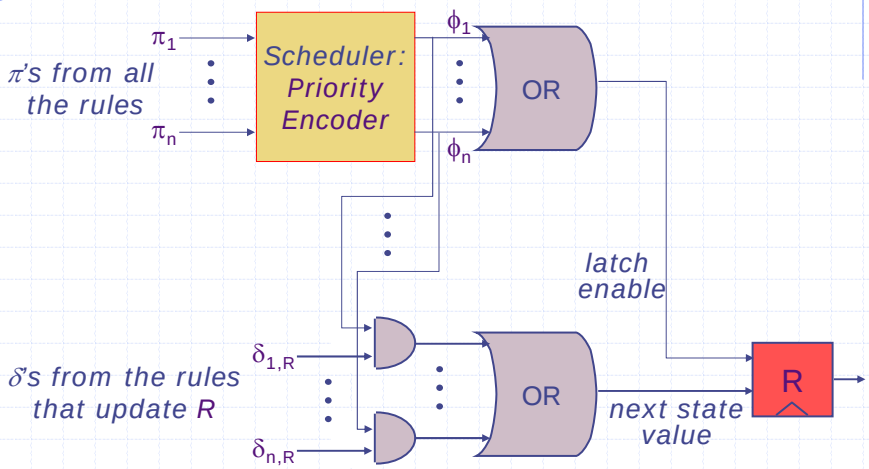


Combining State Updates: *strawman*



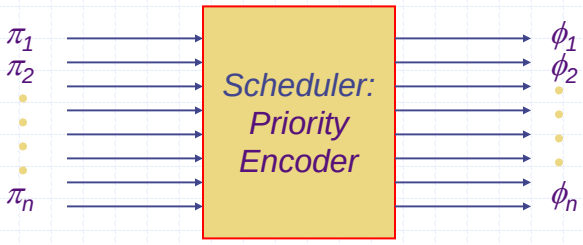
What if more than one rule is enabled?

Combining State Updates



Scheduler ensures that at most one ϕ_i is true

One-rule-at-a-time Scheduler



1. $\phi_i \Rightarrow \pi_i$
2. $\pi_1 \vee \pi_2 \vee .... \vee \pi_n \Rightarrow \phi_1 \vee \phi_2 \vee .... \vee \phi_n$
3. One rewrite at a time
   i.e. at most one ϕ_i is true

Very conservative way of guaranteeing correctness

Executing Multiple Rules Per Cycle

```
rule ra (z > 10);
  x <= x + 1;
endrule

rule rb (z > 20);
  y <= y + 2;
endrule
```

Can these rules be executed simultaneously?

These rules are "conflict free" because they manipulate different parts of the state

$Rule_a$ and $Rule_b$ are conflict-free if

$\forall s \,.\, \pi_a(s) \wedge \pi_b(s) \Rightarrow$

1. $\pi_a(\delta_b(s)) \wedge \pi_b(\delta_a(s))$
2. $\delta_a(\delta_b(s)) == \delta_b(\delta_a(s))$

Executing Multiple Rules Per Cycle

```
rule ra (z > 10);
  x <= y + 1;
endrule

rule rb (z > 20);
  y <= y + 2;
endrule
```

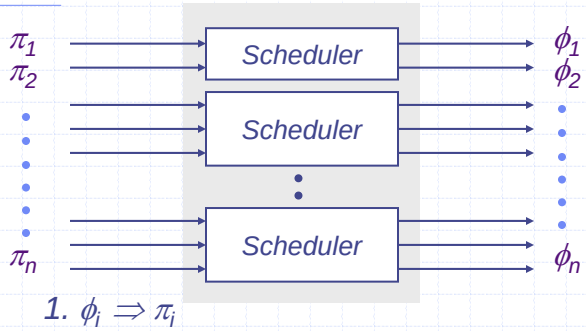
Can these rules be executed simultaneously?

These rules are "sequentially composable", parallel execution behaves like ra < rb

$Rule_a$ and $Rule_b$ are sequentially composable if

$\forall s \,.\, \pi_a(s) \wedge \pi_b(s) \Rightarrow \pi_b(\delta_a(s))$

Multiple-Rules-per-Cycle Scheduler

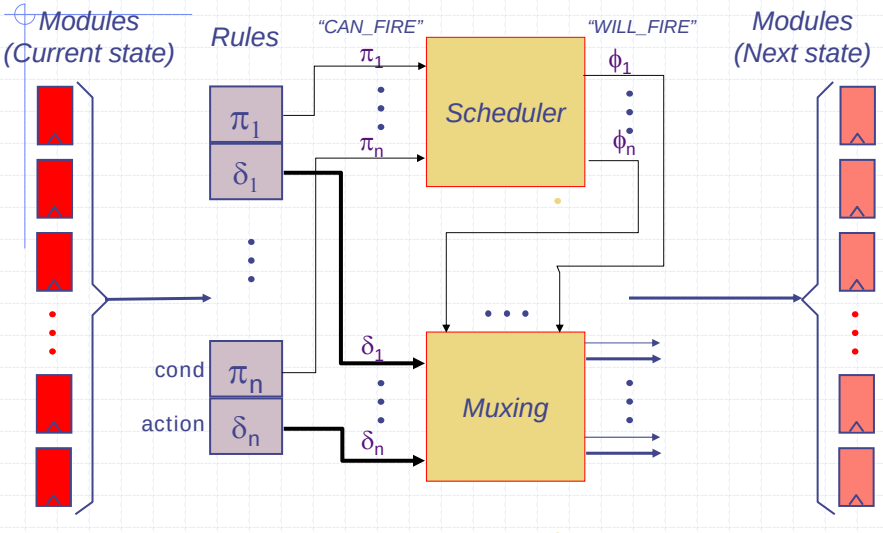


1. $\phi_i \Rightarrow \pi_i$

2. $\pi_1 \vee \pi_2 \vee .... \vee \pi_n \Rightarrow \phi_1 \vee \phi_2 \vee .... \vee \phi_n$

3. Multiple operations such that
   $\phi_i \wedge \phi_j \Rightarrow R_i$ and R_j are conflict-free or sequentially composable

Scheduling and control logic



Synthesis Summary

- Bluespec generates a *combinational hardware scheduler* allowing multiple enabled rules to execute in the same clock cycle
  - The hardware makes a rule-execution decision on every clock (i.e., it is not a static schedule)
  - Among those rules that CAN_FIRE, only a subset WILL_FIRE that is consistent with a Rule order
- Since multiple rules can write to a common piece of state, the compiler introduces suitable muxing and mux control logic
  - This is very simple logic: the compiler will not introduce long paths on its own (details later)

Conditionals and rule-spliting

- In Rule Semantics this rule:

```
rule r1 (p1);
  if (q1) f.enq(x);
  else    g.enq(y);
endrule
```

rule r1 won't fire unless both f anf g queues are not full!

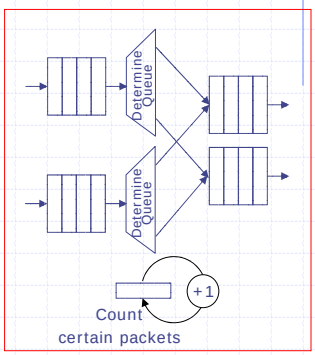
- Is equivalent to the following two rules:

```
rule r1a (p1 && q1);
  f.enq(x);
endrule

rule r1b (p1 && ! q1);
  g.enq(y);
endrule
```

but not quite because of the compiler treats implicit conditions conservatively

Rules for Example 3

```
(* descending_urgency = "r1, r2" *)
// Moving packets from input FIFO i1
rule r1;
  Tin x = i1.first();
  if (dest(x)== 1) o1.enq(x);
  else             o2.enq(x);
  i1.deq();
  if (interesting(x)) c <= c + 1;
endrule

// Moving packets from input FIFO i2
rule r2;
  Tin x = i2.first();
  if (dest(x)== 1) o1.enq(x);
  else             o2.enq(x);
  i2.deq();
  if (interesting(x)) c <= c + 1;
endrule
```



This example won't work properly without rule spliting

Conditionals & Concurrency

```
rule r1a (p1 && q1);
  f.enq(x);
endrule

rule r1b (p1 && ! q1);
  g.enq(y);
endrule
```

```
rule r1 (p1);
  if (q1) f.enq(x);
  else    g.enq(y);
endrule
```

```
rule r2 (p2);
  f.enq(z);
endrule
```

- Suppose there is another rule r2
- Rule r2 cannot be executed simultaneously with r1
  - (conflict on f)
- But rule r2 may be executed simultaneously with r1b
  - (provided p1, !q1 and p2 so permit)
- Thus, splitting a rule can allow more concurrency because of fewer resource conflicts

Scheduling conflicting rules

- When two rules conflict on a shared resource, they cannot both execute in the same clock
- The compiler produces logic that ensures that, when both rules are enabled, only one will fire
- Which one?
  - The compiler chooses
    (and informs you, during compilation)
  - The "descending_urgency" attribute allows the designer to control the choice

Example 2: Concurrent Updates

- Process 0 increments register x;
  Process 1 transfers a unit from register x to register y;
  Process 2 decrements register y


```
rule proc0 (cond0);
    x <= x + 1;
endrule
```

```
rule proc1 (cond1);
    y <= y + 1;
    x <= x – 1;
endrule
```

```
rule proc2 (cond2);
    y <= y – 1;
endrule
```

```
(* descending_urgency = "proc2, proc1, proc0" *)
```

Functionality and performance

- It is often possible to separate the concerns of *functionality* and *performance*
  - First, use Rules to achieve correct functionality
  - *If necessary,* adjust scheduling to achieve application performance goals (latency and bandwidth)*

**I.e., # clocks per datum and # data per clock. Technology performance (clock speed) is a separate issue.*

Improving performance via scheduling

- Latency and bandwidth can be improved by performing more operations in each clock cycle
  - That is, by firing more rules per cycle
- Bluespec schedules all applicable rules in a cycle to execute, except when there are resource conflicts
- Therefore: Improving performance is often about resolving conflicts found by the scheduler

Viewing the schedule

- The command-line flag **`-show-schedule`** can be used to dump the schedule
- Three groups of information:
  - method scheduling information
  - rule scheduling information
  - the static execution order of rules and methods

```
more on this and other rule and method
attributes to be discussed in the Friday
tutorial ...
```


End